Дата последней сборки документа:
12 января 2023 г. в 10:06

Конспект по книге Гудфеллоу «Глубокое обучение»*

Содержание

1 Численные методы **2**

2 Основы машинного обучения **2**

2.1 Точечная оценка . . . 2
2.2 Смещение . . . 2
2.3 Дисперсия . . . 2
2.4 Поиск компромисса между смещением и дисперсией для минимизации среднеквадратической ошибки . . . 3
2.5 Состоятельность . . . 3
2.6 Оценка максимального праводподобия . . . 4
2.7 Метод опорных векторов . . . 5
2.8 Метод главных компонент . . . 5
2.9 Стохастический градиентный спуск . . . 6
2.10 Условное логарифмическое правдоподобие и среднеквадратическая ошибка . . . 7

3 Глубокие сети прямого распространения **7**

3.1 Обучение условных распределений с помощью максимального правдоподобия . . . 7
3.2 Сигмоидные блоки и выходное распределение Бернулли . . . 8
3.3 Блоки softmax и категориальное выходное распределение . . . 8
3.4 Скрытые блоки . . . 8
3.4.1 Блоки линейной ректификации и их обобщения . . . 9
3.4.2 Логистическая сигмоида и гиперболический тангенс . . . 9
3.5 Правило дифференцирования сложной функции . . . 10

4 Регуляризация в глубоком обучении **12**

4.1 Штрафы по норме параметров . . . 12
4.1.1 Регуляризация парамтеров по норме L_2 . . . 12
4.1.2 L_1-регуляризация . . . 14
4.2 Ранняя остановка . . . 15

5 Оптимизация в обучении глубоких моделей **16**

5.1 Чем обучение отличается от чистой оптимизации . . . 16
5.1.1 Минимизация эмпирического риска . . . 17
5.1.2 Суррогатные функции потерь и ранняя остановка . . . 17
5.1.3 Пакетные и мини-пакетные алгоритмы . . . 17
5.2 Проблемы оптимизации нейронных сетей . . . 19
5.2.1 Плохая обусловленность . . . 19

*Гудфеллоу Я., Бенджио И., Курвилль А. Глубокое обучение. – М.: ДМК Пресс, 2018. – 652 с.

1. Численные методы

2. Основы машинного обучения

2.1. Точечная оценка

Точечное оценивание – это попытка найти единственное «наилучшее» предсказание интересующей величины. Пусть $\{x^{(1)}, \ldots, x^{(m)}\}$ – множество m независимых и одинаково распределенных точек. *Точечной оценкой*, или *статистикой*, называется любая функция этих данных

$$\theta_m = g(x^{(1)}, \ldots, x^{(m)}).$$

В этом определении не требуется, чтобы g возвращала значение, близкое к истинному значению θ, ни даже чтобы область значений g совпадала со множеством допустимых значений θ.

Алгоритм k-групповой перекрестной проверки применяется для оценивания ошибки обобщения алгоритма обучения A, когда имеющийся набор данных $\mathbb{D}$ *слишком мал* для того, чтобы простое разделение на обучающий и тестовый или обучающий и контрольный наборы могло дать точную оценку ошибки обобщения, поскольку среднее значение потери L на малом тестовом наборе может иметь высокую дисперсию.

2.2. Смещение

Смещение оценки определяется следующим образом

$$\text{bias}(\hat{\theta}_m) = \mathbb{E}(\hat{\theta}_m) - \theta,$$

где математической ожидание вычисляется по данным (рассматривается как выборка из случайной величины), а θ – истинное значение параметра, которое определяет порождающее распределение.

Оценка $\hat{\theta}$ называется *несмещенной*, если

$$\text{bias}(\hat{\theta}_m) = 0, \text{ т.е. } \mathbb{E}(\hat{\theta}_m) = \theta.$$

Оценка $\hat{\theta}_m$ называется *асимптотически несмещенной*, если

$$\lim_{m\to\infty} \text{bias}(\hat{\theta}_m) = 0, \text{ т.е. } \lim_{m\to\infty} \mathbb{E}(\hat{\theta}_m) = \theta.$$

2.3. Дисперсия

Для определения смещения мы вычисляли математичесвкое ожидание оценки, но точно так же можем вычислить и ее дисперсию. *Дисперсией оценки* называется выражение

$$\text{Var}(\hat{\theta}).$$

Стандартной ошибкой $\text{SE}(\hat{\theta})$ называется квадратный корень из дисперсии.

Воспользовавшись центральной предельной теоремой, согласно которой среднее имеет приблизительно нормальное распределение, можем применить стандартную ошибку для вычисления вероятности того, что истинное математическое ожидание находится в выбранном интервале. Например, *95-процентный доверительный интервал* вокруг выборочного среднего (вокру оценки) $\hat{\mu}_m = \frac{1}{m}\sum_{k=1}^{n} x^{(i)}$ определяется формулой

$$(\hat{\mu}_m - 1.96\,\mathrm{SE}(\hat{\mu}_m), \hat{\mu}_m + 1.96\,\mathrm{SE}(\hat{\mu}_m))$$

при нормальном распределении со средним $\hat{\mu}_m$ и дисперсией $\mathrm{SE}(\hat{\mu}_m)^2$.

NB: В экспериментах по машинному обучению принято говорить, что алгоритм A лучше алгоритма B, если верхняя граница 95-процентного доверительного интервала для ошибки алгоритма A меньше нижней границы 95-процентного доверительного интервала для ошибки алгоритма B.

2.4. Поиск компромисса между смещением и дисперсией для минимизации среднеквадратической ошибки

Что, если имеются две оценки, у одной из которых больше смещение, а у другой дисперсия? Какую выбрать?

Самый распространенный подход к выбору компромиссного решения – воспользоваться *перекрестной проверкой*. Эмпирически продемонстрировано, что перекрестная проверка дает отличные результаты во многих реальных задачах.

Можно также сравнить среднеквартатическую ошибку (MSE) обеих оценок

$$\mathrm{MSE} = \mathbb{E}[(\hat{\theta}_m - \theta)^2] = \mathrm{bias}(\hat{\theta}_m)^2 + \mathrm{Var}(\hat{\theta}_m)$$

Желательной является оценка с малой MSE, именно такие оценки держат под контролем и смещение, и дисперсию. Соотношение между смещением и дисперсией тесно связано с возникающими в машинном обучении понятиями емкости модели, недообучения и переобучения.

Если ошибка обобщения измеряется посредством MSE (и тогда смещение и дисперсия становятся важными компонентами ошибки обобщения), то увеличение емкости (то есть *усложение модели*) влечет за собой *повышение дисперсии* и *снижение смещения*.

2.5. Состоятельность

Обычно нас интересует также поведение оценки по мере роста размера обучающего набора. В частности, мы хотим, чтобы при увеличении числа примеров точечные оценки сходились к истинным значениям соответствующих параметров.

Формально это записывается в виде (условие состоятельности)

$$\hat{\theta}_m \xrightarrow{\mathbf{P}} \theta, \ (m \to \infty)$$

Иногда это условие называют *слабой состоятельностью*, понимая под *сильной состоятельностью* сходимость *почти наверное* $\hat{\theta}$ к θ.

Состоятельность гарантирует, что смещение оценки уменьшается с ростом числа примеров. Однако обратное неверно – из асимптотической несмещенности не вытекает состоятельность.

Рассмотрим, к примеру, оценивание среднего μ нормального распределения $N(x; \mu, \sigma^2)$ по набору данных, содержащему m примеров: $\{x^{(1)}, \ldots, x^{(m)}\}$.

Можно было бы взять в качестве оценки первый пример: $\hat{\theta} = x^{(i)}$. В таком случае $\mathbb{E}(\hat{\theta})_m = \theta$, поэтому оценка является несмещенной вне зависимости от того, сколько примеров мы видели. Отсюда, конечно, следует, что оценка асимптотически несмещенная. Но она не является состоятельной, т.к. *неверно*, что $\hat{\theta}_m \to \theta,\ (m \to \infty)$.

2.6. Оценка максимального праводподобия

Рассмотрим множества m примеров $\mathbb{X} = \{x^{(1)}, \ldots, x^{(m)}\}$, независимо выбираемых из неизвестного порождающего распределения $p_{data}(x)$.

Обозначим $p_{model}(x; \theta)$ параметрическое семейство распределений вероятности над одним и тем же пространством, индексированное параметром θ.

Тогда оценка максимального правдоподобия для θ определяется формулой

$$\theta_{ML} = \arg\max_{\theta} p_{model}(\mathbb{X}; \theta) = \arg\max_{\theta} \prod_{i=1}^{m} p_{model}(x^{(i)}; \theta)$$

Такое произведение большого числа вероятностей по ряду причин может быть неудобно. Например, оно подвержено *потере значимости*. Для получения эквивалентной, но более удобной задачи оптимизации заметим, что взятие логарифма правдоподобия не изменяет arg max, но преобразует произведение в сумму

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{m} \log p_{model}(x^{(i)}; \theta)$$

Поскольку arg max не изменяется при умножении функции стоимости на константу, мы можем разделить правую часть на m и получить выражение в виде математического ожидания относительно эмпирического распределения $\hat{p}_{data}$, определяемого обучающими данными

$$\theta_{ML} = \arg\max_{\theta} \mathbb{E}_{x \sim \hat{p}_{data}} [\log p_{model}(x; \theta)]$$

Один из способов интерпретации оценки максимального правдоподобия состоит в том, чтобы рассматривать ее как минимизацию дивергенции (расхождения) Кульбака-Лейблера между этими эмпирическим распределением $\hat{p}_{data}$, определяемым обучающим набором, и модельным распределением.

Дивергенция Кульбака-Лейблера определяется формулой

$$D_{KL}(\hat{p}_{data} \,||\, p_{model}) = \mathbb{E}_{x \sim \hat{p}_{data}} [\log \hat{p}_{data}(x) - \log p_{model}(x)]$$

Первый член разности в квадратных скобках зависит только от порождающего данные процесса, но не от модели. Следовательно, при обучении модели, минимизирующей дивергенцию КЛ, мы должны минимизировать только величину

$$-\mathbb{E}_{x \sim \hat{p}_{data}} [\log p_{model}(x)],$$

а это, конечно, то же самое, что максимизация величины $\theta_{ML} = \arg\max_{\theta} \mathbb{E}_{x \sim \hat{p}_{data}} [\log p_{model}(x; \theta)]$.

NB: То есть, другими словами задача максимизации правдоподобия эквивалентна задаче минимизации дивергенции Кульбака-Лейблера между эмпирическим распределением $\hat{p}_{data}$ и модельным распределением p_{model}.

2.7. Метод опорных векторов

Линейную функцию в методе опорных векторов можно переписать в виде

$$w^T x + b = b + \sum_{i=1}^{m} \alpha_i x^T x^{(i)},$$

где $x^{(i)}$ – обучающий пример, α – вектор коэффициентов.

Записав алгоритм обучения в таком виде, мы сможем заменить x результатом заданной функции признаков $\varphi(x)$, а скалярное произведение – функцией $k(x, x^{(i)}) = \varphi(x) \cdot \varphi(x^{(i)})$, которая называется ядром.

Заменив скалярное произведение вычислением ядра, мы можем делать предсказание, пользуясь функцией

$$f(x) = b + \sum_{i} \alpha_i k(x, x^{(i)})$$

Основанная на ядре функция в точности эквивалентна предварительной обработке путем применения $\varphi(x)$ ко всем входным данным с последующим обучением линейной модели в новом преобразованном пространстве.

NB: Трюк с ядром полезен по двум причинам:

- Во-первых, он позволяет обучать модели, *нелинейно* зависящие от x, применяя методы выпуклой оптимизации, о которых точно известно, что они сходятся эффективно
- Во-вторых, *ядерная функция* k часто допускает реализацию, значительно *более эффективную с вычислительной точки зрения*, чем наивное построение двух векторов $\varphi(x)$ и явное вычисление их скалярного произведения

Главный недостаток ядерных методов – тот факт, что сложность вычисления решающей функции линейно зависит от числа обучающих примеров, поскольку i-ый пример вносит член $\alpha_i k(x, x^{(i)})$ в решающую функцию.

В методе опорных векторов эта проблема сглаживается тем, что обучаемый вектор α содержит в основном нули. Тогда для классификации нового примера *требуется вычислить ядерную функцию только для обучающих примеров с ненулевыми* α_i. Эти обучающие примеры и называются опорными векторами.

2.8. Метод главных компонент

Метод главных компонент находит ортогональное линейное преобразование, переводящее входные данные x в представление z.

Рассмотрим матрицу плана X размера $m \times n$. Будем предполагать, что матемаческое ожидание данных $\mathbb{E}[x] = 0$. Если это не так, центрирования легко добиться, вычтя среднее из всех примеров на этапе предварительной обработки.

Несмещенная выборочная ковариационная матрица, ассоциированная с X, определяется по формуле

$$Var[x] = \frac{1}{m-1} X^T X$$

PCA находит представление (посредством линейного преобразования) $z = W^T x$, для которого $Var[z]$ – *диагональная*.

Главные компоненты можно получить с помощью сингулярного разложения. Точнее, это правые сингулярные верктоpы. Чтобы убедиться в этом, предположим, что W – правые сингулярные векторы в разложении $X = U\Sigma W^T$. Тогда исходное уравнение собственных векторов можно переписать в базисе W

$$X^T X = (U\Sigma W^T)^T U\Sigma W^T = W\Sigma^2 W^T$$

Разложение SVD полезно для доказательства того, что PCA приводит к диагональной матрице $Var[z]$. Применяя сингулярное разложение X, мы можем выразить дисперсию X в виде

$$Var[x] = \frac{1}{m-1} X^T X = \frac{1}{m-1} (U\Sigma^2 W^T)^T U\Sigma W^T = \frac{1}{m-1} W\Sigma^2 W^T,$$

где используется тот факт, что $U^T U = I$, поскольку матрца U в сингулярном разложении по определению ортогональная. Отсюда следует, что ковариационная матрица z диагональная

$$Var[z] = \frac{1}{m-1} Z^T Z = \frac{1}{m-1} W^T X^T X W = \frac{1}{m-1} W^T W\Sigma^2 W^T W = \frac{1}{m-1} \Sigma^2$$

На этот раз мы воспользовались тем, что $W^T W = I$ – опять же по определению сингулярного разложения.

Проведенный анализ показывает, что представление, полученное в результате проецирования данных x на z посредством линейного преобразования W, имеет *диагональную ковариационную матрицу* Σ^2. А отсюда сразу вытекает, что *взаимная корреляция отдельных элементов z равна нулю*.

2.9. Стохастический градиентный спуск

Стохастический градиентный спуск (СГС) имеет важные применения и за пределами глубокого обучения. Это основной способ обучения *больших линейных моделей* на очень больших наборах данных. Для модели фиксированноо размера стоимость одного шага СГС не зависит от размера обучающего набора m. Количество шагов до достижения сходимости обычно возрастает с ростом размера обучающего набора. Но когда m стремится к бесконечности, модель в итоге сходится к наилучшей возможной ошибке тестирования еще до того, как СГС проверил каждый пример из обучающего набора. Дальнейшее увеличение m не увеличивает время обучения, необходимое для достижения наилучшей ошибки тестирования. С этой точки зрения можно сказать, что асимптотическая стоимость обучения модели методом СГС как функции от m имеет порядок $O(1)$.

2.10. Условное логарифмическое правдоподобие и среднеквадратическая ошибка

Линейную регрессию можно интерпретировать как *нахождение оценки максимального правдоподобия*. Будем считать, что цель не в том, чтобы вернуть одно предсказание $\hat{y}$, а чтобы построить модель, порождающую условное распределение $p(y|\mathbf{x})$. Цель алгоритма обучения теперь – аппроксимировать $p(y|\mathbf{x})$, подогнав его под все эти разные значения y, совместимые с $\mathbf{x}$.

Для вывода такого же алгоритма линейной регрессии, как и раньше, определим

$$p(y|\mathbf{x}) = N(y; \hat{y}(\mathbf{x}, \mathbf{w}), \sigma^2)$$

Функкция $\hat{y}(\mathbf{x}; \mathbf{w})$ дает предсказание среднего значения нормального распределения. В этом примере мы предполагаем, что дисперсия фиксирована и равна константе σ^2. Поскольку предполагается, что примеры независимы и одинаково распределены, то условное логарифмическое правдоподобие записывается в виде

$$\sum_{i=1}^{m} \log p(y^{(i)}|\mathbf{x}^{(i)}; \theta) = -m \log \sigma - \frac{m}{2} \log 2\pi - \sum_{i=1}^{m} \frac{\|\hat{y}^{(i)} - y^{(i)}\|^2}{2\sigma^2},$$

где $\hat{y}^{(i)}$ – результат линейной регрессии для i-ого примера $\mathbf{x}^{(i)}$, а m – число обучающих примеров.

Сравнивая логарифмическое правдоподобие со среднеквадратической ошибкой

$$\text{MSE}_{train} = \frac{1}{m} \sum_{i=1}^{m} \|\hat{y}^{(i)} - y^{(i)}\|^2,$$

мы сразу же видим, что *максимизация логарифмического правдоподобия* относительно $\mathbf{w}$ дает ту же оценку параметров $\mathbf{w}$, что *минимизация среднеквадратической ошибки*.

Значения этих критериев различны, но положение оптимума совпадает. Это служит обоснованием использования среднеквадратической ошибки в качестве оценки максимального правдоподобия.

3. Глубокие сети прямого распространения

3.1. Обучение условных распределений с помощью максимального правдоподобия

Большинство современных нейронных сетей обучается *с помощью максимального правдоподобия*. Это означает, что в качестве функции стоимости берется отрицательное логарифмическое правдоподобие, которое можно эквивалентно описать как перекрестную энтропию между обучающими данными и распределением модели

$$J(\theta) = -\mathbb{E}_{x,y \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{y}|\mathbf{x})$$

Одно необычное свойство перекрестной энтропии, используемой при вычислении оценки максимального правдоподобия, заключается в том, что для типичных встречающихся на практике моделей у нее, как правило, нет минимального значения. Если выходная величина дискретна, то в большинстве моделей параметризация устроена так, что модель неспособна представить

вероятность 0 или 1, но может подойти к ней сколь угодно близко. Примером может служить логистическая регрессия.

3.2. Сигмоидные блоки и выходное распределение Бернулли

Во многих задачах требуется предсказывать значение бинарной величины y. В таком виде можно представить задачу классификации с двумя классами.

Подход на основе максимального правдоподобия заключается в определении распределения Бернулли величины y при условии $\mathbf{x}$.

Градиент $\mathbf{0}$ обычно приводит к проблемам, потому что у алгоритма обучения нет никаких указаний на то, как улучшить параметры.

Лучше применять другой подход, который гарантирует, что градиент обязательно будет достаточно большим, если модель дает неверный ответ. Этот подход основан на использовании сигмоидных выходных блоков в сочетании с максимальным правдоподобием.

Сигмоидный выходной блок

$$\hat{y} = \sigma(\mathbf{w}^T\mathbf{h} + b),$$

где σ – логистическая сигмоида.

3.3. Блоки softmax и категориальное выходное распределение

Если требуется представить распределение вероятности дискретной случайной величины, принимающей n значений, то можно воспользоваться функцией softmax. Ее можно рассматривать как обобщение сигмоиды, которая использовалась для представления распределения бинарной величины.

Функция softmax чаще всего используется как выход классификатора для представления распределения вероятности n классов. Реже функция softmax используется внутри самой модели.

Как и сигмоида, функция активации softmax склонна к насыщению. У сигмоиды всего один выход, и она насыщается, когда абсолютная величина аргумента очень велика. У softmax выходных значений несколько. Они насыщаются, когда велика абсолютная величина разностей между входными значениями.

softmax – это сглаженный вариант $\arg\max$. Сложность спектрального разложения матрицы $d \times d$ имеет порядок $O(d^3)$.

3.4. Скрытые блоки

Некоторые скрытые блоки, не являются всюду дифференцируемыми. Например, функция линейной ректификации (ReLU) $g(z) = \max\{0, z\}$ не дифференцируема в точке $z = 0$. Может показаться, что из-за этого g непригодна для работы с алгоритмами обучения градиентными методами. Но на практике градиентный спуск работает для таких моделей машинного обучения достаточно хорошо. Отчасти это связано с тем, что алгоритмы обучения нейронных сетей обычно не достигают локального минимума функции стоимости, а просто находят достаточно малое значение. Поскольку мы не ожидаем, что обучение выйдет на точку, где градиент равен 0, то можно смириться с тем, что минимум функции стоимости соответствует точкам, в которых градиент не определен. Недифференцируемые скрытые блоки обычно не дифференцируемы лишь в немногих

точках. В общем случае функцию $g(z)$ имеет производную слева, определяемую коэффициентом наклона функции слева от z, и аналогично производную справа. Функция дифференцируема в точке z, только если производные слева и справа определены и равным между собой. Для функции $g(z) = \max\{0, z\}$ производная слева в точке $z = 0$ равна 0, а производная справа равна 1. В программных реализациях обучения нейронной сети обычно возвращается какая-то односторонняя производная, а не сообщается, что производная не определена и не возбуждается исключение. Эвристически это можно оправдать, заметив, что градиентная оптимизация на цифровом компьютере в любом случае подвержена численным погрешностям. Когда мы просим вычислить $g(0)$, крайне маловероятно, что истинное значение действительно равно 0. Скорее всего, это какое-то малое значение, округленное до 0.

Важно, что на практике можно *спокойно игнорировать недифференцируемость функции активации* скрытых блоков.

3.4.1. Блоки линейной ректификации и их обобщения

В блоке линейной ректификации используется функция активации $g(z) = \max\{0, z\}$. Эти блоки легко оптимизировать, потому что они очень похожи на линейные. Разница только в том, что блок линейной ректификации в половине своей области определения выводит 0. Поэтому производная блока линейной ректификации остается большой всюду, *где блок активен*.

Блоки линейной ректификации обычно применяются *после* аффинного преобразования

$$h = g(W^T x + b)$$

При инициализации параметров аффинного преобразования рекомендуется присваивать всем элементам b небольшое положительное значение, например, 0.1. Тогда блок линейной ректификации в начальный момент с большой вероятностью окажется активен для большинства обучающих примеров, и производная будет отлична от нуля.

Недостатком блоков линейной ректификации является невозможность обучить их градиентными методами на примерах, для которых функция активации блока равна нулю. Различные обобщения гарантируют, что градиент имеется в любой точке.

Три обобщения блоков линейной ректификации основаны на использовании ненулевого углового коэффициента α_i, когда $z_i < 0 : h_i = \max(0, z_i) + \alpha_i \min(0, z_i)$.

В случае абсолютной ректификации берутся фиксированные значения $\alpha_i = -1$, так что $g(z) = |z|$. Такая функция активации используется при распознавании объектов в изображении, где имеет смысл искать признаки, инвариантные относительно изменения полярности освещения. Другие обобщения находят более широкие применения. В случае ReLU с утечкой α_i принимаются равными фиксированному малому значению, например, 0.01, а в случае параметрического ReLU α_i считается обучаемым параметром.

Блоки линейной ректификации и все их обобщения основаны на принципе, согласно которому модель проще обучить, если ее поведение близко к линейному.

3.4.2. Логистическая сигмоида и гиперболический тангенс

Блоки линейной ректификации стали использоваться сравнительно недавно, а раньше большинство нейронных сетей в роли функции активации применялась логистическая сигмоида или гиперболический тангенс.

Сигмоидные блоки в качестве выходных предсказывают вероятность того, что бинарная величина равна 1. В отличие от кусочно-линейных, сигмоидные блоки близки к асимптоте в большей части своей области определения – приближаются к высокому значению, когда z стремится к бесконечности, и к низкому, когда z стремится к минус бесконечности. Высокой чувствительностью они обладают только в окрестности нуля. Из-за *насыщения* сигмоидальных блоков *градиентное обучение сильно затруднено*. Поэтому использование их в качестве скрытых блоков в сетях прямого распространения ныне не рекомендуется.

Если использовать сигмоидальную функцию активации необходимо, то лучше взять не логистическую сигмоиду, а гиперболический тангенс. Он ближе к тождественной функции в том смысле, что $\tanh(0) = 0$, тогда как $\sigma(0) = 1/2$.

Поскольку tanh походит на тождественную функцию в окрестности нуля, обучение глубокой нейронной сети $\hat{y} = w^T \tanh(U^T \tanh(V^T x))$ напоминает обучение линейной модели $\hat{y} = w^T U^T V^T x$, при условии, что сигналы активации сети удается удерживать на низком уровне. При этом обучение сети с функцией активации tanh упрощается.

Сигмоидальные функции активации все же применяются, но не в сетях прямого распространения. К реккурнетным сетям, многим вероятностным моделям и некоторым автокодировщикам предъявляются дополнительные требования, исключающие использование кусочно-линейных функций.

3.5. Правило дифференцирования сложной функции

Это правило применяется для вычисления производных функций, являющихся композициями других функций, чьи производные известны.

Пусть $\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n$, g отображает $\mathbb{R}^m$ в $\mathbb{R}^n$, а f отображает $\mathbb{R}^n$ в $\mathbb{R}$. Тогда правило дифференцирования сложной функции в векторной форме запишется в виде

$$\nabla_x z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_y z,$$

где $\partial \mathbf{y}/\partial \mathbf{x}$ – *матрица Якоби* функции g размера $n \times m$.

Отсюда видно, что градиент по переменной $\mathbf{x}$ можно получить, умножив матрицу Якоби $\partial \mathbf{y}/\partial \mathbf{x}$ на градиент $\nabla_y z$.

Обычно алгоритм обратного распространения применяется к тензорам произвольной размерности, а не просто к векторам. Концептуально это то же самое, что применение к векторам. Разница только в том, как числа организуются в сетке для представления тензора. Можно вообразить, что тензор сериализуется в вектор перед обратным распространением, затем вычисляется векторозначный градиент, после чего градиент снова преобразуется в тензор.

В таком переупорядоченном представлении обратное распространение – это все то же умножение якобиана на градиенты.

Для обозначения градиента значения z относительно тензора $\mathbf{X}$ мы пишем $\nabla_{\mathbf{X}} z$, как если бы $\mathbf{X}$ был просто вектором. Теперь индексы элементов $\mathbf{X}$ составные – например, трехмерный тензор индексируется тремя координатами. Мы можем абстрагироваться от этого различия, считая, что одна переменная i представляет целый кортеж индексов. Для любого возможного индексного кортежа i $(\nabla_{\mathbf{X}} z)_i$ обозначает частную производную $\partial z/\partial \mathbf{X}_i$ – точно так же, как для любого целого индекса i $(\nabla_x z)_i$ обозначает $\partial z/\partial x_i$. В этих обозначениях можно записать правило диф-

ференцирования сложной функции в применении к тензорам. Если $\mathbf{Y} = g(\mathbf{X})$ и $z = f(\mathbf{Y})$, то

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial Y_j}$$

Алгоритм обратного распространения был разработан, чтобы избежать многократного вычисления одного и того же выражения при дифференцировании сложной функции. Из-за таких повторов время выполнения наивного алгоритма могло расти экспоненциально. Теперь, можно оценить вычислительную сложность алгоритма обратного распространения.

Если предположить, что стоимость вычисления всех операций приблизительно одинакова, то вычислительную сложность можно проанализировать в терминах количества выполненных операций. Следует помнить, что под единицей мы понимаем базовую единицу графа вычислений, в действительности она может состоять из нескольких арифметических операций (например, умножение матриц может считаться одной операцией в графе). Вычисление градиента в графе с n вершинами никогда не приводит к выполнению или сохранению результатов более $O(n^2)$. Здесь мы подсчитываем в графе вычислений, а не отдельные аппаратные операции, поэтому важно понимать, что время выполнения разных операций может значительно различаться.

Легко видеть, что для вычисления градиента требуется не более $O(n^2)$ операций, потому что на этапе прямого распространения в худшем случае будут обсчитаны все n вершин исходного графа (в зависимости от того, какие значения мы хотим вычислить, может потребоваться обойти весь граф).

Поскольку граф вычислений – это ориентированный ациклический граф, число ребер в нем не более $O(n^2)$. Для типичных графов, встречающихся на практике, ситуация даже лучше. В большинстве нейронных сетей функции стоимости имеют в основном цепную структуру, так что сложность обратного распространения равна $O(n)$. Это намного лучше, чем наивный подход, при котором число обрабатываемых вершин иногда растет *экспоненциально*!

Откуда возникает экспоненциальный рост, можно понять, раскрыв и переписав правило дифференцирования сложной функции без рекурсии (здесь сумма по путям $u^{(\pi_1, u^{(\pi_2)}), \ldots, u^{(\pi_t)}}$ из $\pi_1 = j$ в $\pi_t = n$)

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum \prod_{k=2}^{t} \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}$$

Поскольку количество путей из вершины j в вершину n может экспоненциально зависит от длины пути, то число слагаемых в этой сумме, равное числу таких путей, может расти экспоненциально с увеличением глубины графа прямого распространения. Такая высокая сложность связана с многократным вычислением $\partial u^{(i)} / \partial u^{(j)}$. Чтобы *избежать повторных вычислений*, мы можем рассматривать *обратное распространение как алгоритм заполнения таблицы*, в которой храняться промежуточные результаты $\partial u^{(n)} / \partial u^{(i)}$.

Каждой вершине графа соответствует элемент таблицы, в котором хранится градиент для этой вершины. Заполняя таблицу в определенном порядке, алгоритм обратного распространения избегает повторного вычисления многих ошибок подвыражений. Такую стратегию заполнения таблицы иногда называют динамическим программированием.

4. Регуляризация в глубоком обучении

4.1. Штрафы по норме параметров

Многие подходы к регляризации основаны на ограничении емкости моделей путем прибавления штрафа по норме параметра $\Omega(\theta)$ к целевой функции J

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta),$$

где $\alpha \in [0, \infty]$ – гиперпараметр, задающий вес члена Ω, штрафующего по норме, относительно стандратной целевой функции J. Чем больше значение α, тем сильнее регуляризация.

В нейронных сетях мы обычно предпочитаем штрафовать по норме *только веса* аффинного преобразования в каждом слое, оставляя смещения нерегуляризованными.

Регуляризация параметров смещения может стать причиной значительного недообучения.

4.1.1. Регуляризация парамтеров по норме L_2

Полная целевая функция с регуляризацией Тихонова имеет вид

$$\tilde{J}(w; X, y) = \frac{\alpha}{2} w^T w + J(w; X, y),$$

а градиент по параметрам

$$\nabla_w \tilde{J}(w; X, y) = \alpha w + \nabla_w J(w; X, y)$$

Один шаг обновления весов с целью уменьшения градиента имеет вид

$$w \leftarrow w - \varepsilon \left(\alpha w + \nabla_w J(w; X, y)\right)$$

То же самое можно переписать в виде

$$w \leftarrow (1 - \varepsilon\alpha) w - \varepsilon \nabla_w J(w; X, y)$$

Как видим, добавление члена сложения весов изменило правило обучения: теперь мы на каждом шаге умножаем вектор весов на постоянный коэффициент, меньший 1, перед тем как выполнить стандартное обновление градиента.

Еще упростим анализ, предположив квадратичную аппроксимацию целевой функции в окрестности того значения весов, при котором достигается минимальная стоимость обучения без регуляризации. Если целевая функция действительно квадратичная, как в случае модели линейной регрессии со среднеквадратической ошибкой, то такая аппроксимация идеальна. Аппроксимация $\tilde{J}$ описывается формулой

$$\hat{J}(\theta) = J(w^*) + 1/2(w - w^*)^T H (w - w^*),$$

где H – матрица Гессе J относительно w, вычисленная в точке w^*. В этой квадратичной аппроксимации нет члена первого порядка, потому что w^*, по определению, точка минимума, в которой градиент обращается в нуль.

Минимум $\hat{J}$ достигается там, где градиент

$$\nabla_w \hat{J}(w) = H(w - w^*) = 0$$

Чтобы изучить эффект снижения весов, прибавим градиент снижения весов. Теперь мы можем найти из него минимум регуляризованного варианта $\hat{J}$. Обозначим $\tilde{w}$ положение точки минимума.

$$\begin{aligned} \alpha\tilde{w} + H(\tilde{w} - w^*) &= 0 \\ (H + \alpha I)\tilde{w} &= Hw^* \\ \tilde{w} &= (H + \alpha I)^{-1} Hw^* \end{aligned}$$

Поскольку матрица H вещественная и симметричная, мы можем разложить ее в произведение диагональной матрицы Λ и ортогональной матрицы собственных векторов Q

$$H = Q\Lambda Q^T$$

Тогда

$$\tilde{w} = (Q\Lambda Q^T + \alpha I)^{-1} Q\Lambda Q^T w^* = Q(\Lambda + \alpha I)^{-1}\Lambda Q^T w^*.$$

Компонента w^*, параллельная i-ому собственному вектору H, умножается на коэффициент $\lambda_i/(\lambda_i + \alpha)$. Вдоль направлений, для которых собственные значения H относительно велики, например, когда $\lambda_i \gg \alpha$, эффект регуляризации сравнительно мал. Те же компоненты, для которых $\lambda_i \ll \alpha$, сжимаются почти до нуля.

Если направление не дает вклада в уменьшение целевой функции, то собственное значение гессиана мало, т.е. движение в этом направялении не приводит к заметному возрастнию градиента. Компоненты вектора весов, соответствующие таким малозначным направлениям, снижаются почти до нуля благодаря использованию регуляризации в ходе обучения.

Рассмотрим линейную регрессию, в которой истинная функция стоимости квадратическая. Для линейной регрессии функция стоимости равна сумме квадратов ошибок

$$(xw - y)^T (Xw - y)$$

После добавления L_2-регуляризации целевая функция принимает вид

$$(Xw - y)^T (Xw - y) + \frac{1}{2} w^T w$$

В результате *нормальные уравнения*, из которых ищется решение

$$w = (X^T X)^{-1} X^T y$$

принимают вид

$$w = (X^T X + \alpha I)^{-1} X^T y$$

Матрица $X^T X$ пропорциональна *ковариационной матрице* $1/m X^T X$. Применение L_2-регуляризации заменяет эту матрицу на $(X^T X + \alpha I)^{-1}$. Новая матрица отличается от исходной только прибавлением α ко всем диагональным элементам. *Диагональные элементы* этой матрицы соответствуют *дисперсии каждого входного признака.*

Таким образом, *L_2-регуляризация* заставляет алгоритмы обучения «воспринимать» вход X *как имеющий более высокую дисперсию* и, следовательно, уменьшать веса тех признаков, для которых ковариация с выходными метками мала, по сравнению с добавленной дисперсией.

NB: L_2-регуляризация занижает веса признаков, которые обнаруживают слабую ковариацию с целевой меткой.

4.1.2. L_1-регуляризация

Формально L_1-регуляризация параметров модели w определяется по формуле

$$\Omega(\theta) = \|w\|_1 = \sum_i |w_i|,$$

т.е. как сумма абсолютных величин отдельных параметров.

Регуляризованная целевая функция описывается формулой

$$\tilde{J}(w; X, y) = \alpha \|w\|_1 + J(w; X, y),$$

а ее градиент (точнее, *частичный градиент*) равен

$$\nabla_w \tilde{J}(w; X, y) = \alpha \operatorname{sign}(w) + \nabla_w J(w; X, y),$$

где $\operatorname{sign}(w)$ означает, что функция sign применяется к каждому элементу w.

Теперь вклад регуляризации в градиент уже не масштабируется линейно с ростом каждого w_i, а описывается постоянным слагаемым, знак которого совпадает с $\operatorname{sign}(w_i)$. Одним из следствий является тот факт, что мы уже не получим изящных алгебраических выражений квадратичной аппроксимации $J(X; y, w)$, как в случае L_2-регуляризации.

Квадратичную аппроксимацию L_1-регуляризованной целевой функции можно представить в виде суммы по параметрам

$$\hat{J}(w; X, y) = J(w; X, y) + \sum_i \left[\frac{1}{2} H_{i,i} (w_i - w_i^*)^2 + \alpha |w_i| \right]$$

У задачи минимизации этой приближенной функции стоимости имеется аналитическое решение (для каждого измерения i) вида

$$w_i = \operatorname{sign}(w_i^*) \max \left[|w_i^*| - \frac{\alpha}{H_{i,i}}, 0 \right]$$

Предполагается, что матрица Гессе диагональная $H = \operatorname{diag}([H_{1,1}, \ldots, H_{n,n}])$, где все $H_{i,i} > 0$.

Предположим, что $w_i^* > 0$ для всех i. Тогда есть два случая:

1. $w_i^* \leqslant \alpha / H_{i,i}$. Тогда оптимальное значение w_i для регуляризованной целевой функции будет просто $w_i = 0$,
2. $w_i^* > \dfrac{\alpha}{H_{i,i}}$. Тогда регуляризация не сдвигает оптимальное значение w_i в нуль, а просто смещает его в этом направлении на расстояние $\alpha / H_{i,i}$.

Аналогичное рассуждение проходит, когда $w_i^* < 0$, только L_1-штраф увеличивает w_i на $\alpha/H_{i,i}$ или обращает в 0.

По сравнению с L_2-регуляризацией, L_1-регуляризация дает более *разреженное* решение. В этом контексте под разреженностью понимается тот факт, что у некоторых параметров оптимальное значение равно 0.

Свойство разреженности, присущее L_1-регуляризации, активно эксплуатировалось как механизм отбора признаков, идея которого состоит в том, чтобы упростить задачу машинного обучения за счет выбора некоторого подмножества располагаемых признаков. В частности, хорошо известная модель LASSO (Least Absolute Shrinkage and Selection Operator) объединяет L_1-штраф с линейной моделью и среднеквадратической функцией стоимости. Благодаря L_1-штрафу некоторые веса обращаются в 0, и соответствующие им признаки отбрасываются.

Многие *стратегии регуляризации* можно интерпретировать как *байесовский вывод* на основе *оценки апостериорного максимума* (MAP). В частности, L_2-регуляризация эквивалента байесовскому выводу на основе MAP с *априорным нормальным распределением весов*. В случае L_1-регуляризации штраф $\alpha\Omega(w) = \alpha \sum_i |w_i|$, применяемый для регуляризации функции стоимости, эквивалентен члену, содержащему логарифм априорного расределения, который максимизируется байесовским выводом на основе MAP, когда в качестве *априорного* используется *изотропное распределение Лапласа векторов* $w \in \mathbb{R}^n$.

4.2. Ранняя остановка

При обучении больших моделей, репрезентативная емкость которых достаточно для переобучения, мы часто наблюдаем, что ошибка монотонно убывает со временем, но ошибка на контрольном наборе снова начинает расти.

Это означает, что мы могли бы получить модель с более низкой ошибкой на *контрольном* наборе (и, хочется надеятся, на *тестовом* тоже), вернувшись к тем значениям параметров, которые существовали на момент наименьшей ошибки. Всякий раз как ошибка на контрольном наборе улучшается, мы сохраняем копию параметров модели. Когда алгоритм обучения завершается, мы возвращаем именно эти, а не самые последние параметры. А завершается алгоритм, когда на протяжении заранее заданного числа итераций не удается улучшить параметры, по сравнению с наилучшим запомненным ранее. Эта стратегия называется *раней остановкой*. Пожалуй, это самая распространенная *форма регуляризации* в машинном обучении.

К дополнительным расходам на раннюю остановку следует также отнести хранение копии наилучших параметров. Вообще говоря, эти расходы пренебрежимо малы, поскольку параметры можно хранить в медленной памяти большого объема (например, обучение производится в памяти GPU, а оптимальные параметры хранятся в памяти хоста-компьютера или на диске). Поскольку оптимальные параметры записываются сравнительно редко и никогда не читаются в процессе обучения, такие нечастые операции записи слабо сказываются на общем времени обчения.

Раняя остановка – ненавязчивая форма регуляризации в том смысле, что не требуется вносить почти никаких изменений в базовую процедуру обучения, целевую функцию или множество допустимых значений параметров. Следовательно, раннюю остановку можно легко использовать, не изменяя динамику обучения. Совершенно не так обстоит дело со снижением весов, когда нужно внимально следить за тем, чтобы не снизить веса слишком сильно и не завести сеть в плохой локальный минимум, соответвующий патологически малым весам.

Раннюю остановку можно использовать автономно или в сочетании с другими стратегиями регуляризации. Для ранней остановки необходим контрольный набор, а значит, часть обучающих данных не следует подавать на вход модели.

Каким образом ранняя остановка выступает в роли регуляризатора? В работах Bishop (1995) и Sjoberg and Ljung (1995) утверждается, что благодаря ранней остановке процедура оптимизации ограничивается просмотром сравнительно небольшой области пространства параметров в окрестности начального значения параметров θ_0. Точнее, допустим, что выбрано τ шагов оптимизации (что соответствует τ *итерациям обучения*) и *скорость обучения* ε. Произведение $\varepsilon\tau$ можно рассматривать как меру эффективной емкости. В предположении, что градиент ограничен, наложение ограничений на число итераций и скорость обучения лимитируют область пространства параметров, достижимую из θ_0. В этом смысле $\varepsilon\tau$ ведет себя как величина, обратная коэффициенту снижения весов.

Действительно, можно показать, что в случае простой модели линейной регрессии с квадратической функцией ошибки и обычного градиентного спуска ранняя остановка эквивалентна L_2-регуляризации.

Тогда

$$\tau \approx \frac{1}{\varepsilon\alpha} \text{ и } \alpha \approx \frac{1}{\tau\varepsilon}.$$

Таким образом, в этих предположениях число итераций обучения τ играет роль величины, обратно пропорциональной параметру L_2-регуляризации, а число, обратное $\varepsilon\tau$, – роль коэффициента снижения весов.

Значения параметров, соответвующие направлениям сильной кривизны целевой функции, регуляризируются меньше, чем в направлениях меньшей кривизны.

5. Оптимизация в обучении глубоких моделей

5.1. Чем обучение отличается от чистой оптимизации

Алгоритмы оптимизации, используемые для обучения глубоких моделей, отличаются от традиционных алгоритмов оптимизации в нескольких отношениях. Машинное обучение обычно работает не напрямую. В большинстве ситуаций нас интересует некоторая мера качества P, которая определена относительно тестового набора и может оказаться вычислительно неприступной. Поэтому мы оптимизируем P косвенно. Мы уменьшаем другую функцию стоимости $J(\theta)$ в надежде, что при этом улучшится и P. Это резко отличается от чистой оптимизации, где минимизация J и есть конечная цель.

Типичную функцию стоимости можно представить в виде среднего по обучающему набору

$$J(\theta) = \mathbb{E}_{(\mathbf{x},y)\sim\hat{p}_{data}} L(f(\mathbf{x};\theta), y),$$

где L – функция потерь на одном примере, $f(x;\theta)$ – предсказанный выход для входа x, $\hat{p}_{data}$ – *эмпирическое* распределение. В случае обучения с учителем y – ассоциированная с входом метка.

Это уравнение определяет целевую функцию относительно обучающего набора.

Но мы обычно предпочитаем минимизировать соответсвующую целевую функцию, в которой математическое ожидание берется по *порождающему данные распределению* p_{data}, а не просто по

конечному обучающему набору

$$J^*(\theta) = \mathbb{E}_{(\mathbf{x},y)\sim p_{data}} L(f(\mathbf{x};\theta), y). \tag{1}$$

5.1.1. Минимизация эмпирического риска

Цель алгоритма машинного обучения – уменьшить математическое ожидание ошибки обобщения, описываемое формулой (1). Эта величина называется *риском*. Подчеркнем еще раз, что математическое ожидание берется по истинному распределению p_{data}. Если бы мы знали истинное распределение $p_{data}(\mathbf{x}, y)$, то минимизация риска была бы задачей оптимизации, решаемой с помощью алгоритма оптимизации. Но когда $p_{data}(\mathbf{x}, y)$ неизвестно, а есть только обучающий набор примеров, мы имеем задачу машинного обучения.

Простейший способ преобразовать задачу машинного обучения в задачу оптимизации – минимизировать ожидаемые потери на обучающем наборе. Это значит, что мы заменяем истинное распределение $p(x, y)$ эмпирическим распределением $\hat{p}(x, y)$, определяемым по обучающему набору. И теперь требуется минимизировать *эмпирический риск*

$$\mathbb{E}_{(x,y)\sim\hat{p}_{data}} L(f(x;\theta), y) = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta), y^{(i)}),$$

где m – количество обучающих примеров.

Тем не менее минимизация эмпирического риска уязвима для переобучения. Модели высокой емкости могут попросту запомнить обучающий набор. Во многих случаях минимизация эмпирического риска практически неосуществима. На практике используется подход, при котором фактически оптимизируемая величина еще сильнее отличается от той, которую мы хотели бы оптимизировать на самом деле.

5.1.2. Суррогатные функции потерь и ранняя остановка

Типичное условие ранней остановки основано на контрольном наборе, и предназначено для того, чтобы остановить работу алгоритма, когда возникает угроза переобучения. Обучение зачастую заканчивается, когда производные суррогатной функции потерь все еще велики, и этим разительно отличается от чистой оптимизации, при которой считается, что алгоритм сошелся, если градиент стал очень малым.

5.1.3. Пакетные и мини-пакетные алгоритмы

Большинство свойств целевой функции J, используемой чуть ли не во всех наших алгоритмах оптимизации, также выражается *в терминах математического ожидания по обучающему набору*. Например, чаще всего используется ее градиент

$$\nabla_\theta J(\theta) = \mathbb{E}_{x,y\sim\hat{p}_{data}} \nabla_\theta \log p_{model}(x, y; \theta)$$

Вычисление точного значения этого математического ожидания обошлось бы очень дорого, потому что для этого нужно вычислить модель на каждом примере из набора данных. На практике можно *случайно* выбрать небольшое число примеров и *усреднить* только по ним.

Еще одно сообщение в пользу статистического оценивания градиента по небольшой выборке связано с избыточностью обучающего набора. В худшем случае все m примеров в обучающем

наборе в точности совпадают. Оценка градиента по выборке дала бы правильное значение, взяв всего один пример, т.е. было бы затрачено в m раз меньше времени, чем при наивном подходе. На практике нам вряд ил встретится худший случай, но все же можно найти много примеров, дающих очень похожий вклад в градиент.

Алгоритмы оптимизации, в которых используется весь обучающий пакет, называются *пакетными градиентными методами*, поскольку обрабатывают сразу все примеры одним большим пакетом. Как правило, термин «пакетный градиентный спуск» подразумевает использование всего обучающего набора.

Алгоритмы оптимизации, в которых используется *по одному примеру за раз*, иногда называют *стохастическими методами*.

Большинство алгоритмов, используемых в глубоком обучении, находятся где-то посередине – число примеров в них больше одного, но меньше размера обучающего набора. Традиционно они назывались мини-пакетными, а сейчас – просто стохастическими.

Канонический пример стохастического метода – стохастический градиентный спуск.

На размер мини-пакета оказывают влияние следующие факторы:

- чем больше пакет, тем точнее оценка градиента, но зависимость хуже линейной,
- если пакет очень мал, то не удается в полной мере задействовать преимущества многоядерной архитектуры. Поэтому существует никий абсолютный минимум размера пакета – такой, что обработка мини-пакетов меньшего размера не дает никакого выигрыша во времени,
- если все примеры из пакета нужно обрабатывать параллельно (так обычно и бывает), то размер пакета лимитирован объемом памяти. Для многих аппаратных конфигураций размер пакета – ограничивающий фактор,
- для некоторых видов оборудования оптимальное время выполнения достигается при определенных размерах массива. Так, для GPU наилучшие результаты получаются, когда размер пакета – степень 2. Типичный пакет имеет размер от 32 до 256, а для особо больших моделей иногда пробуют 16,
- небольшие пакеты могут дать эффект регуляризации, быть может, из-за шума, который они вносят в процесс обучения.

Методы, которые вычисляют обновления только на основе градиента g, обычно сравнительно устойчивы и могут работать с пакетами небольшого размера, порядка 100. Методы второго порядка, в которых используется также матрица Гессе H и которые вычисляют такие обновления, как $H^{-1}g$, обычно нуждаются в пакетах гораздо большего размера, порядка 10 000. Такие большие пакеты нужны, чтобы свести к минимуму флуктуации в оценках $H^{-1}g$.

Важно также, чтобы мини-пакеты выбирались *случайно*. Для вычисления *несмещенной оценки ожидаемого градиента по выборке* необходимо, чтобы примеры были *независимы*. Мы также хотим, чтобы две последовательные *оценки градиента* были *независимы* друг от друга, поэтому два последовательных мини-пакета примеров тоже должны быть *независимы*.

Многие наборы данных естественно упорядочены так, что между последовательными примерами имеется высокая корреляция. Например, длинный список результатов анализа крови, скорее всего, организован так, что сначала идут пять анализов одного пациента, взятых в разные моменты времени, затем – три анализа второго пациента и т.д. Если бы мы выбирали примеры из такого набора, то каждый мини-пакет оказался бы очень сильно смещенным, т.к. представлял бы преимущественно одного пациента из многих присутствующих в наборе данных.

В тех случаях, когда порядок примеров в наборе не случаен, *необходимо перетасовать пакет, прежде чем формировать мини-пакеты.* Для очень больших наборов, насчитывающих миллиарды примеров, выбирать примеры по-настоящему случайно при каждом построении мини-пакета не всегда возможно. К счастью, на практике обычно достаточно перетасовать набор один раз и затем хранить его в таком виде. При этом получается фиксированный набор возможных мини-пакетов последовательных примеров, которым вынуждены будут пользоваться все обучаемые впоследствии модели, и каждая модель будет видеть примеры в одном и том же порядке при проходе по обучающим данным. Но похоже, что такое отклонение от истинно случайного выбора не оказывает значимого негативного эффекта. Тогда как полное пренебрежение перетасовкой примеров способно серьезно снизить эффективность алгоритма.

В большинстве реализаций мини-пакетного алгоритма стохастического градиентного спуска набор данных перетасовывается один раз, после чего по нему производится несколько проходов. На первом проходе каждый мини-пакет используется для вычисления несмещенной оценки истинной ошибки обобщения. На втором проходе оценка становится смещенной, потому что получена повторной выборкой уже использованных значений, а не новых примеров из порождающего распределения.

Тот факт, что алгоритм стохастического градиентного спуска действительно минимизирует ошибку обобщения, отчетливее всего виден в онлайновом обучении, когда примеры или мини-пакеты выбираются из потока данных. Иными словами, обучаемая модель не получает обучающего набора фиксированного размера, а, подобно живому существу, в каждый момент времени видит новый пример; при этом каждый пример $(x; y)$ поступает из порождающего распределения $p_{data}(x, y)$. В такой ситуации примеры никогда не повторяются, каждое испытание – честная выборка из p_{data}.

5.2. Проблемы оптимизации нейронных сетей

Традиционно в машинном обучении избегали сложностей общей оптимизации за счет тщательного выбора целевой функции и ограничений, *гарантирующих выпуклость* задачи оптимизации. При обучении нейронных сетей приходится сталкиваться с общим невыпуклым случаем. Но даже выпуклая оптимизация не обходится без сложностей.

5.2.1. Плохая обусловленность

Ряд проблем возникает даже при оптимизации выпуклых функций. Самая известная из них – *плохая обусловленность матрицы Гессе* H. Это очень общая проблема, присущая большинству методов численной оптимизации, все равно, выпуклой или нет.

Отсупление: задача оказалась плохо обусловленной. Сравнительно небольшие возмущения системы уравнений привели к существенным отклонениям в решении. Обусловленность задачи не связана с конкретным численным методом, это *неустранимая ошибка.* Существуют способы снижения погрешности, вызванной плохой обусловленностью: 1) каким-то образом перейти к хорошо обусловленной эквивалентной системе, 2) повысить точность определения коэффициентов СЛАУ и правой части.

Плохо обусловленные системы являются обобщением понятия вырожденных систем. Системы «близкие» к *вырожденным* скорее всего будут *плохо обусловлены.*

Число $\mu(A) \equiv \|A\| \cdot \|A^{-1}\|$ называется *числом обусловленности матрицы* и дает универсальную оценку относительной погрешности решения системы с матрицей A

$$\frac{\|\delta x\|}{\|x\|} \leqslant \mu(A) \frac{\|\delta f\|}{\|f\|}$$

какой бы ни была правая часть f.

Число обусловленности $\mu(A)$ зависит от выбранной матричной нормы. Например, для евклидовой нормы $\sigma_{\max}(A^{-1}) = \dfrac{1}{\sigma_{\min}(A)}$ и число обусловленности матрицы в евклидовой норме принимает вид

$$\mu_e(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}.$$

Евклидова норма матрицы A равна ее максимальному сингулярному числу

$$\|A\|_e = \sigma_{\max}(A) = \sqrt{\lambda_{\max}(A^T A)}$$

Для симметричных матриц $A = A^T$ сингулярные числа σ совпадают с модулями собственных значений $|\lambda|$, а сингулярные векторы совпадают с собственными векторами.

Поскольку любая норма матрицы не меньше ее наибольшего по модулю собственного значения, то $\|A\| \geqslant \max|\lambda A|$; поскольку собственные значения матриц A и A^{-1} взаимо обратны, то

$$\mu(A) \geqslant \frac{\max|\lambda_A|}{\min|\lambda_A|} \geqslant 1$$

При $\mu \approx 1 \dots 10$ ошибки входных данных слабо сказываются на решении и система считается хоршо обусловленной. При $\mu \gg 10^2 \dots 10^3$ система является плохо обусловленной [3, стр. 38]

Считается, что проблема плохой обусловленности присутствует во всех задачах обучения нейронных сетей. Она может проявляться в «*застревании*» стохастического градиентного спуска в том смысле, что даже очень малые шаги увеличивают функцию стоимости.

Разложение функции стоимости в ряд Тейлора до членов второго порядка показывает, что шаг градиентного спуска величиной $-\varepsilon g$ увеличивает стоимость на

$$\frac{1}{2}\varepsilon^2 g^T H g - \varepsilon g^T g.$$

Плохая обусловленность градиента становится проблемой, когда $1/2\varepsilon^2 g^T H g$ больше $\varepsilon g^T g$. Чтобы понять, страдает ли задача обучения нейронной сети от плохой обусловленности, можно понаблюдать за квадратом нормы градиента $g^T g$ и членом $g^T H g$. Во многих случаях норма градиента не сильно уменьшается за время обучения, тогда как член $g^T H g$ возрастает больше, чем на порядок. В результате обучение происходит очень медленно, несмотря на большой градиент, т.к. приходится уменьшать скорость обучения, чтобы компенсировать еще большую кривизну.

Список литературы

1. *Рамальо Л.* Python – к вершинам мастерства: Лаконичное и эффективное программирование. – М.: МК Пресс, 2022. – 898 с.

2. *Хейдт М., Груздев А.* Изучаем pandas. – М.: ДМК Пресс, 2019. – 682 с.
3. *Петров И.Б.* Лекции по вычислительной математике. – М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2006. – 523 с.